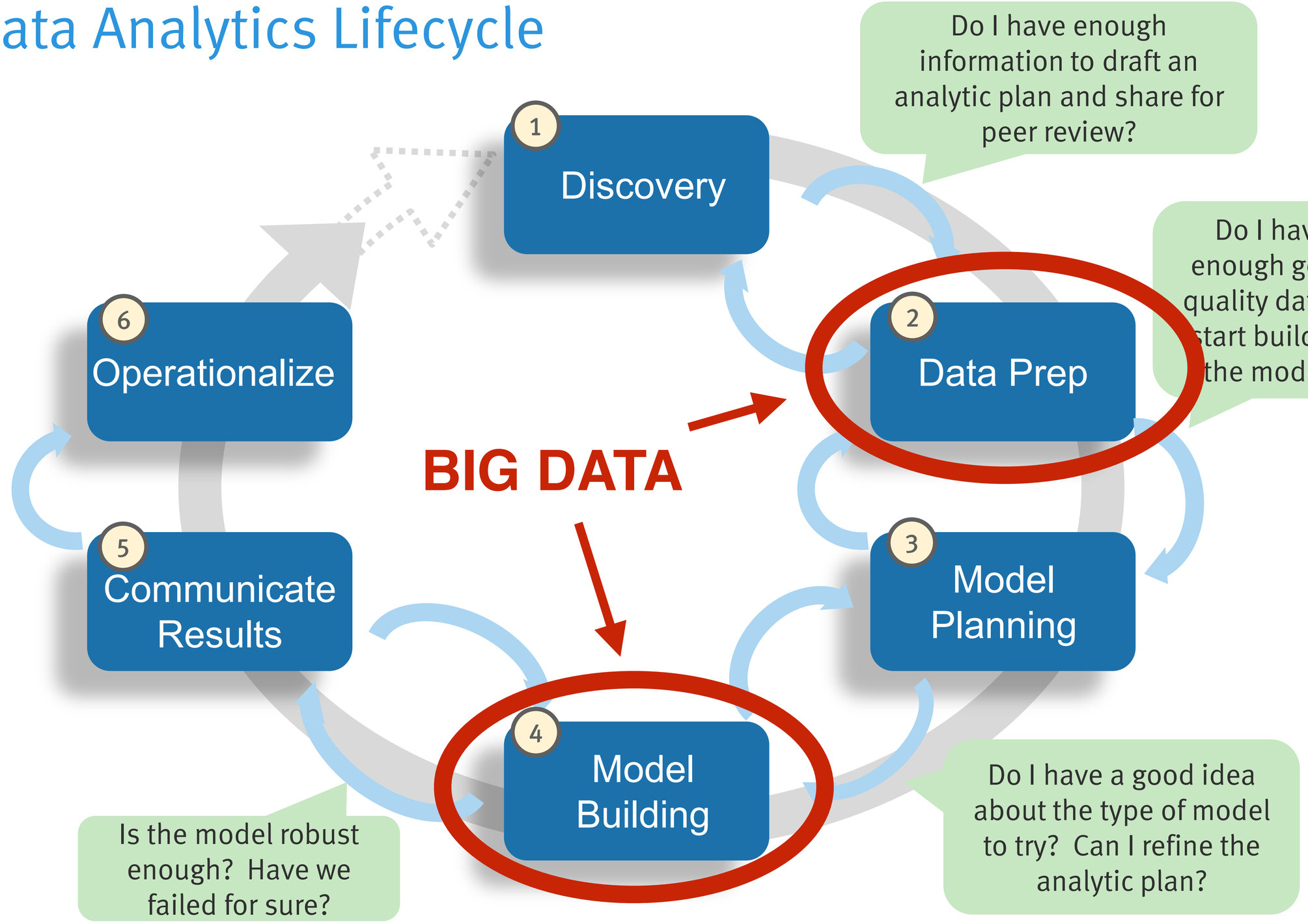
Dealing with Big Data: Streaming Computation Higher-Order Functions

02/02/2017

Huy T. Vo

NYU
CUSP
CENTER FOR URBAN SCIENCE+PROGRESS
CUNY
The City College of New York

Data Analytics Lifecycle



Volume
Velocity
Variety
Veracity
...

[EMC Corporation 2014]

Common Big Data Challenges

- **Volume:**
  - ***Too much data to process*** — reduce processing time by using distributed & parallel computing (next class)
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*
- **Velocity** — ***data comes in real-time*** → cannot store → process **on the fly**!
    - Detecting network failures from inspecting billions of packets per hour

Scaling: dealing with Big Data Volume and Velocity

- **Scale up** — *make the process more scalable on a* **single** *computer*
  - Some distinguishes “scale up” & “scale down” (scaling is a relative term) [Phillip B. Gibbons — IPDPS 2015]
  - ***Scale down*** the amount of data processed or the resources needed to perform the processing
  - ***Scale up*** the computing resources on a node, via parallel processing & faster memory/storage
    - or *scale up* the efficiency/performance of the process!
- **Scale out** — *add more computing resources using* **multiple** *computers*

Common Big Data Challenges

- **Volume:**
  - ***Too much data to process*** — reduce processing time by using distributed & parallel computing (next class)
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Giver **Scale up with Streaming!**
- **Velocity** — ***data comes in real-time*** → cannot store → process **on the fly**!
    - Detecting network failures from inspecting billions of packets per hour

Streaming Algorithms

- An active area of *theory* computation:
  - Study complexity (big O notation), error bounds, correctness, etc.
- Started in the 1970s, but getting more popular thanks to its applications in massive data processing
- Big data world adopts the stream processing model
  - Process data as soon as it arrives (asynchronous updates)
  - Continuous and incremental processing

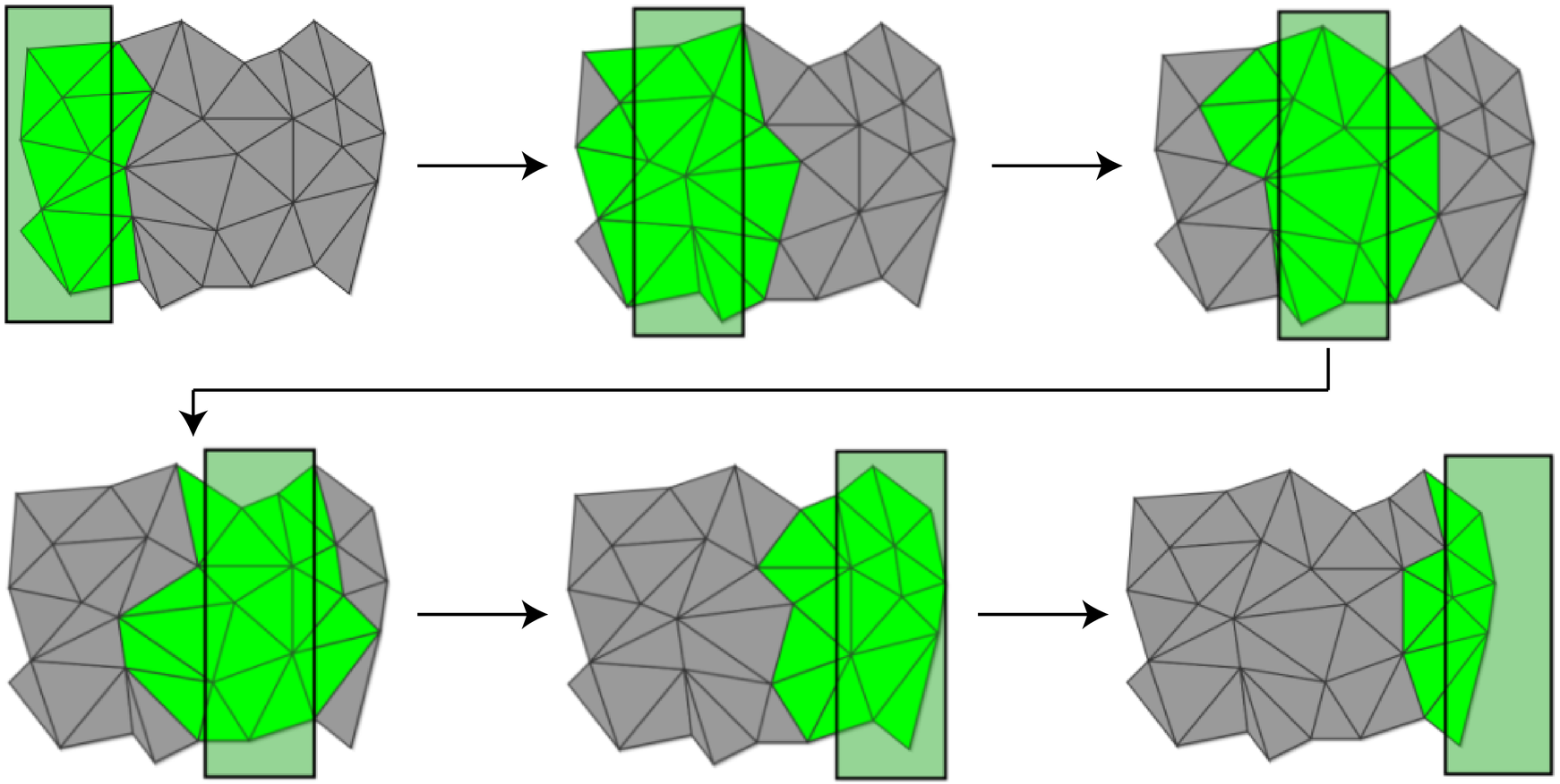
Streaming Computation

- Given a data series of **n** elements: $<a_1, a_2, \ldots, a_n>$ (**n** is usually known) that *can only be examined in* ***a limited number of passes*** (usually one).
- Compute a function of stream, e.g. average, median, distribution (histograms), distinct elements, etc.
- Constraints:
  - **Limited working memory** of size **m** (**m** << **n**)
  - Elements are accessed sequentially (**no random access**)
  - **Fast processing** time per element

How to handle data streams?

- Many representations of the streaming model:
  - Time-Series (most popular in urban data, **storing temporal values**)
  - Cash-Register (mini histogram on data arrivals, **storing counts**)
  - Turnstile (arrival-departure summarization, **storing deltas**)
  - Sliding-window (keep a continuous subset of the input, **storing a buffer**)
- Many classes of techniques to process data elements:
  - Sampling (reduce data inputs)
  - Sketching (data aggregation)
  - Counting (data compression, either lossy or lossless)

Sliding Window



Example: find the missing number

- There are 11 soccer players, whose shirt numbers are from 1 to 11, walking from the tunnel to the field. But only 10 of them arrives.
- How to determine the missing number in the “stream” of players?
- Constraints:
  - We can only *look at a number* at a time
  - We can only *“store” a number* in our head

8 2 6 1 10 3 5 11 9 7

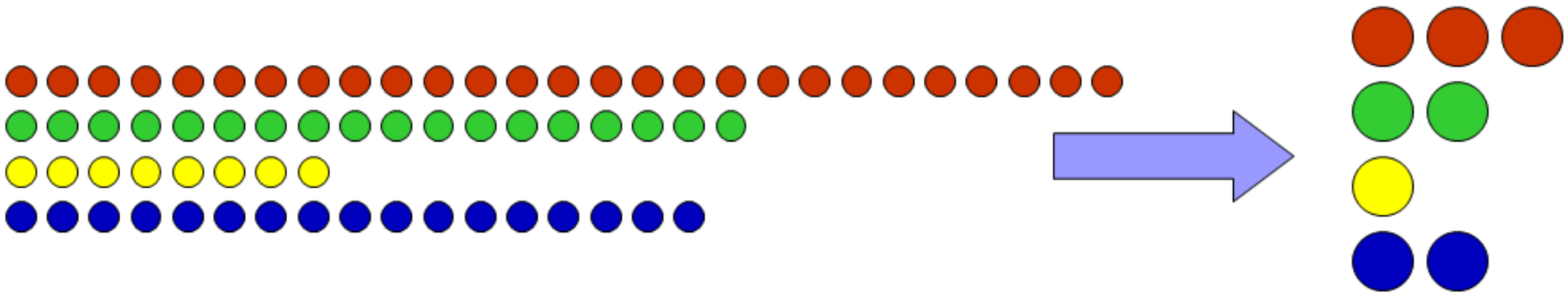
Example: find the missing number

- It's **4**
- The sum of all numbers is fixed: (1+11)*11/2 = **66**
- Record only the sum of the numbers as we scan through the stream
- 8,10,16,17,27,30,35,46,55,**62**
- Our missing number is 66 - 62 = **4**!

8 2 6 1 10 3 5 11 9 7

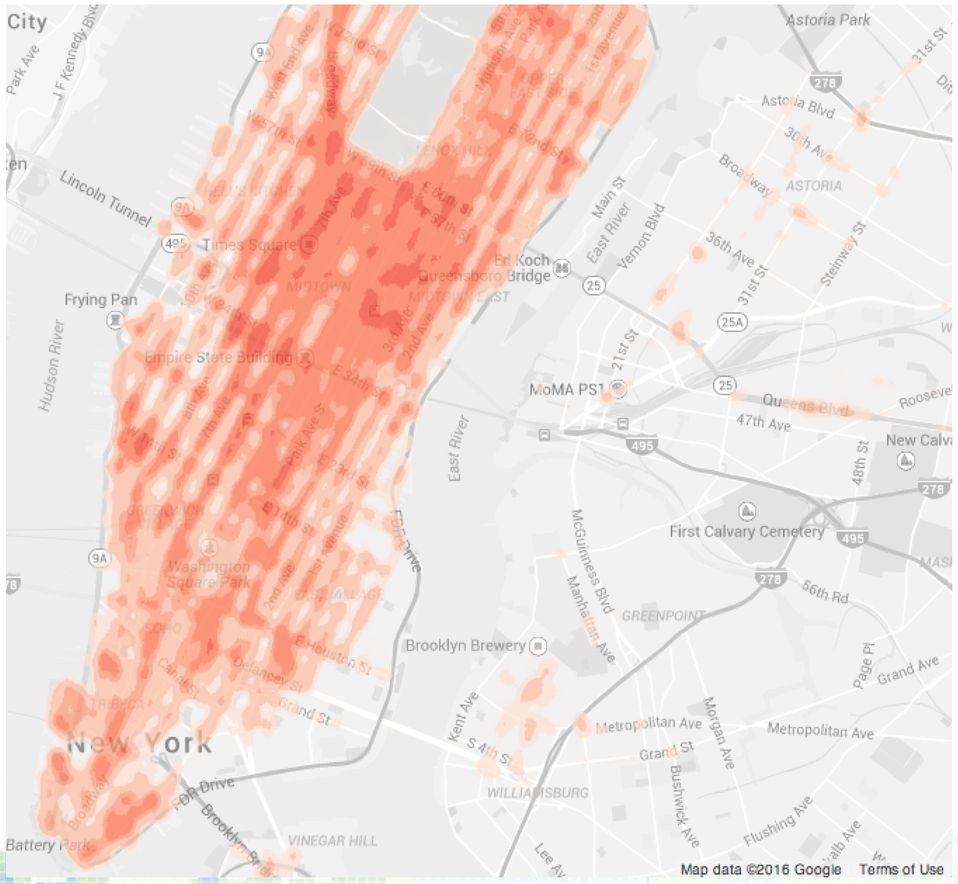
Sampling Techniques

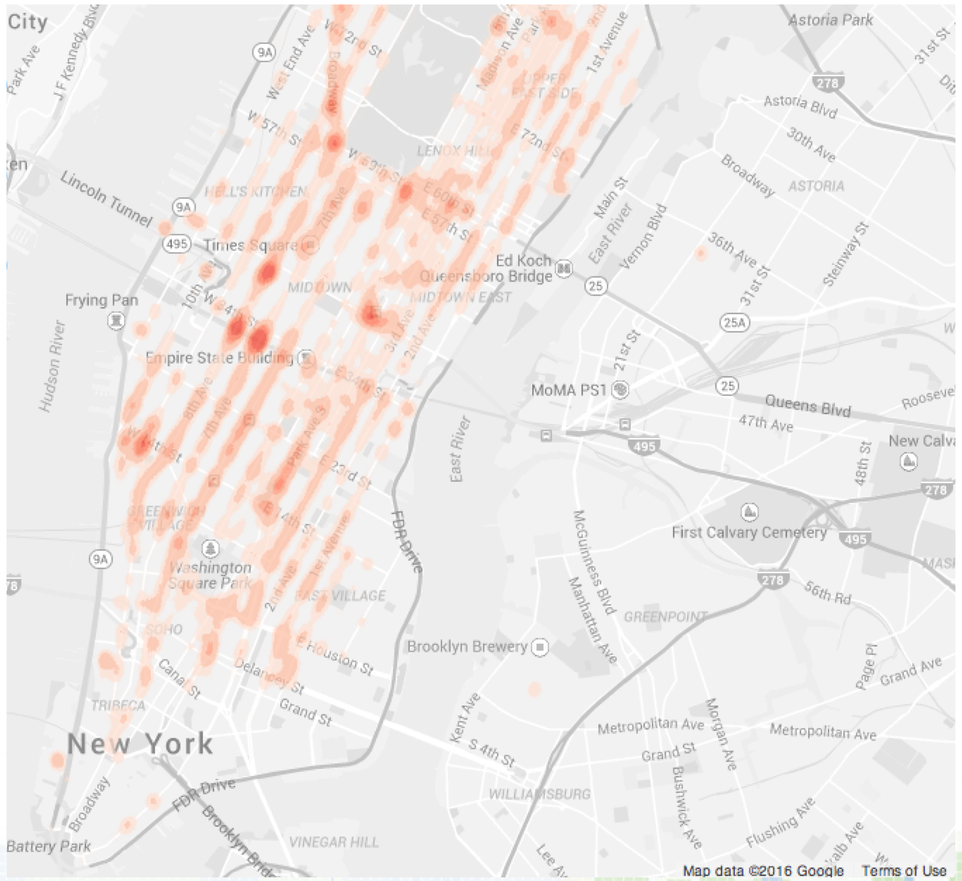
- Motivation: a small random sample of the data can be a good representation of all the data
- Action: sample the data based on some probability model
- A challenge for data streams of unknown size



Sampling Techniques

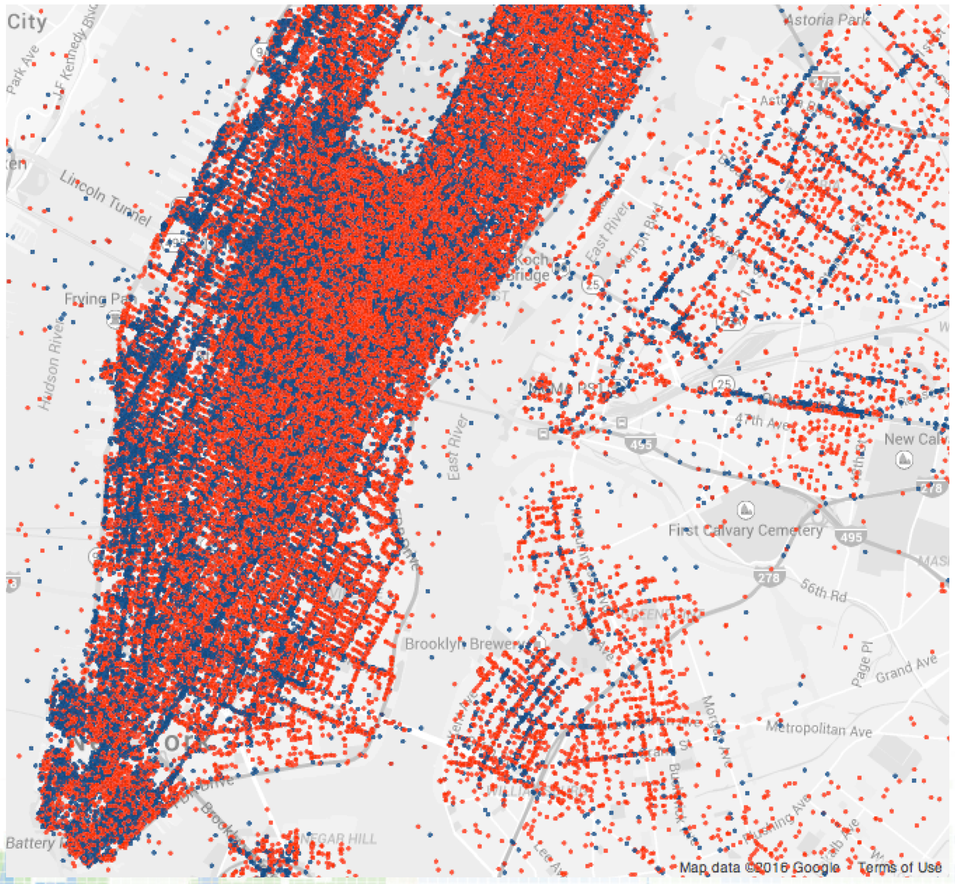
- Mostly acceptable for showing patterns

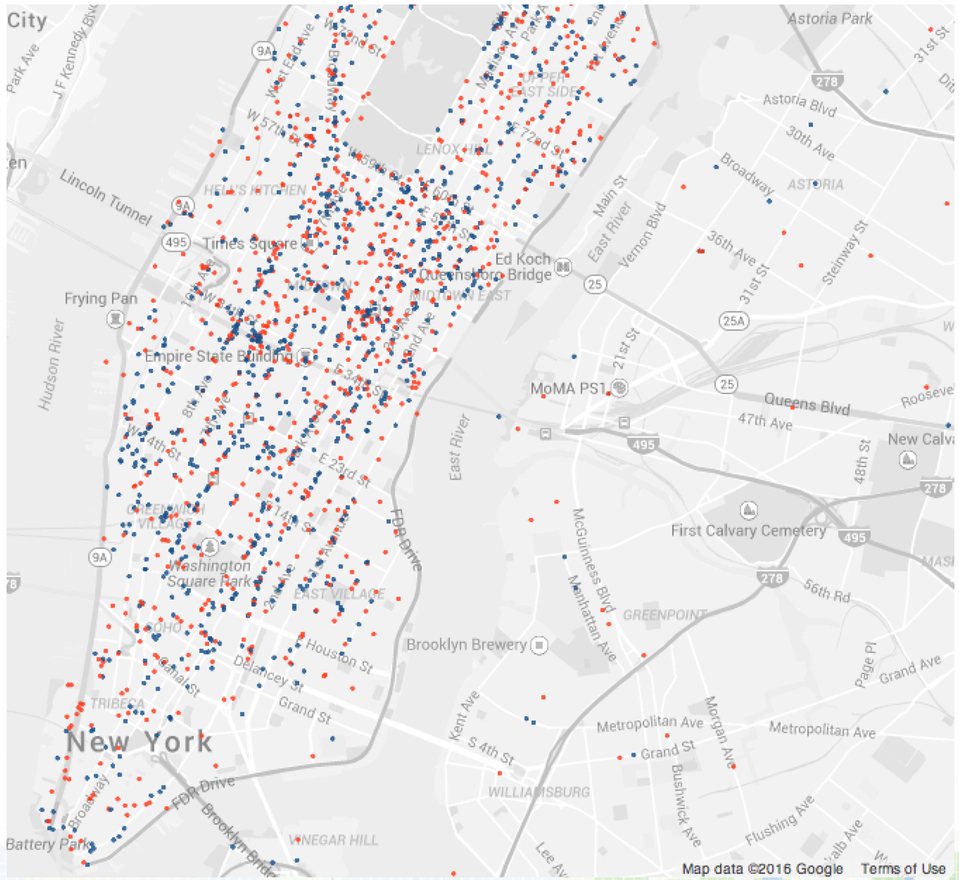




Sampling Techniques

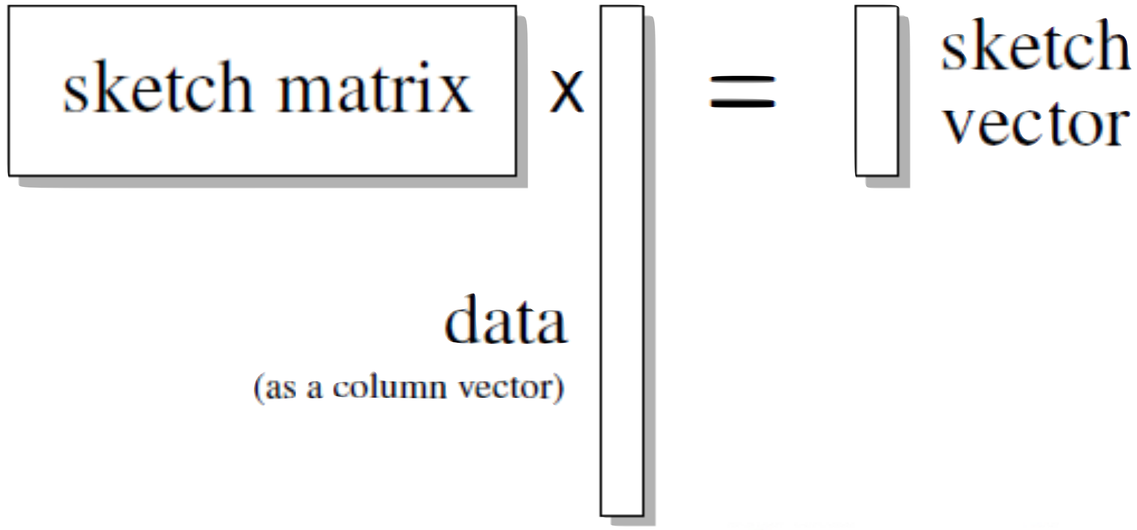
- But not for detecting anomalies!



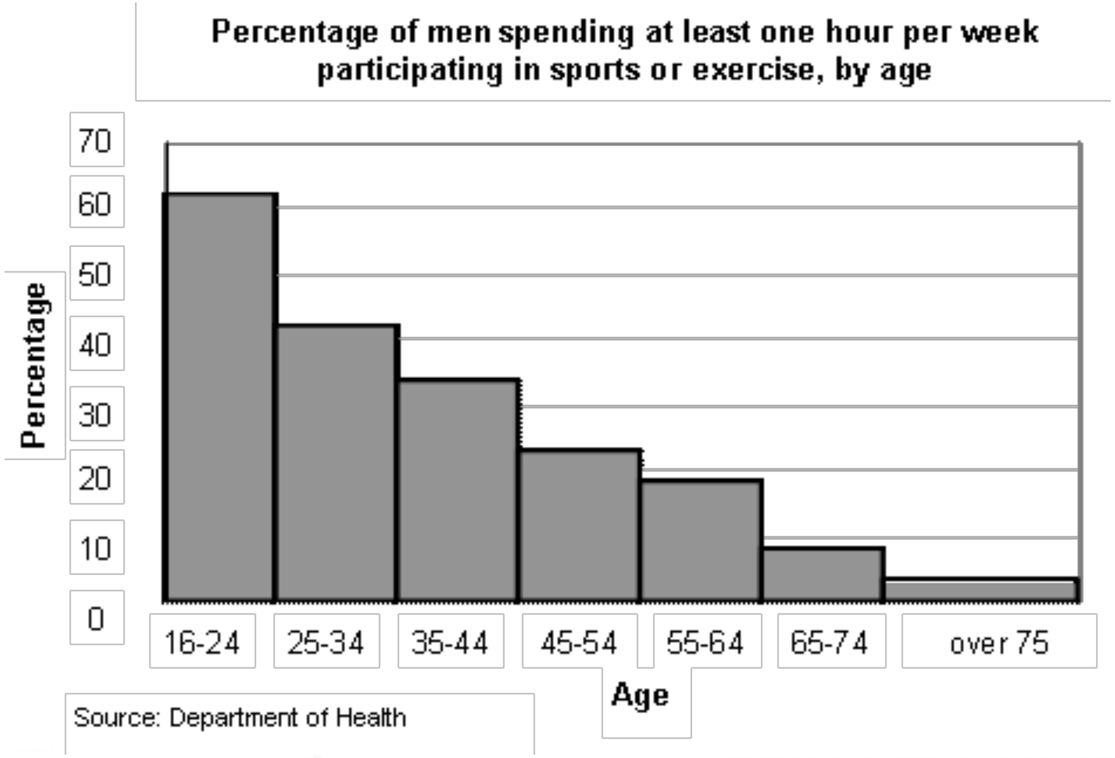


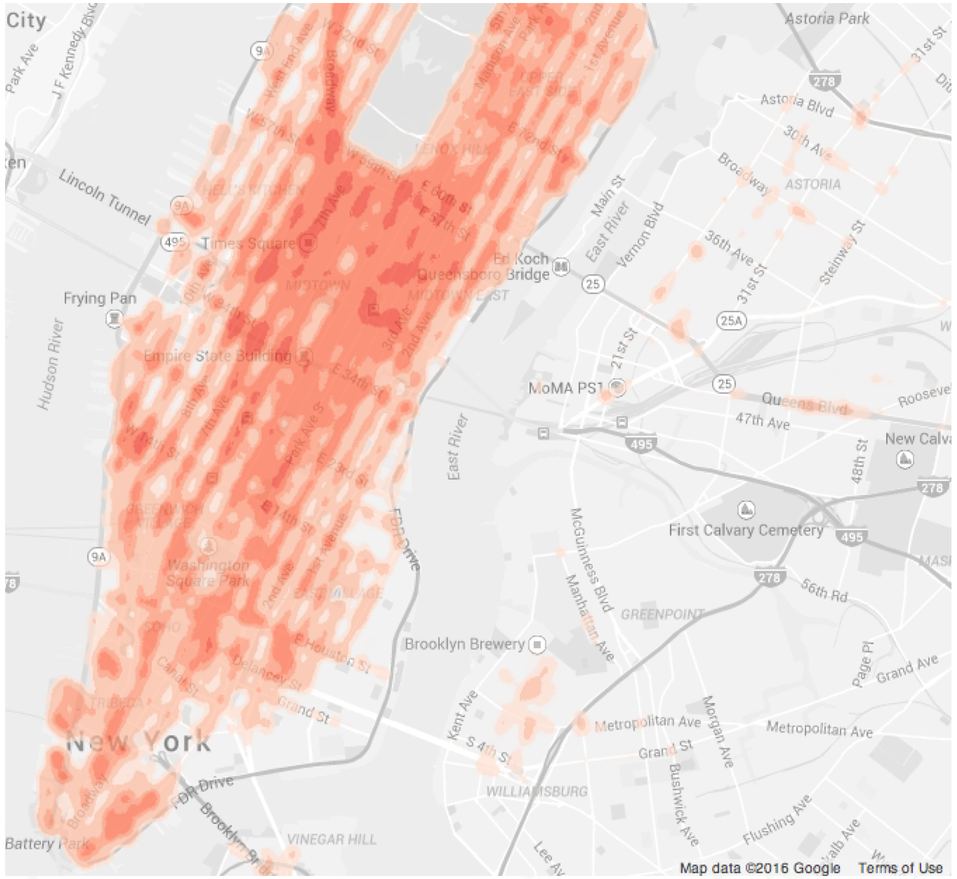
Sketching

- Motivation: only certain pieces or transformation of the data are needed for computing the function of the input stream
- Action: project data into a “sketch” space, progressively build the function of stream
- Accumulation/Aggregation
- Challenge: the sketch could be large



Sketching Examples





Sketching Examples

- Compute a stream average: sketch the (sum, count)
  - For each element, add its value to the sum and increase the count
- Compute a stream histogram: sketch the count per category
  - For each element, add to the count of its category
- Compute a stream heatmap: sketch the “image”
  - For each element, add a KDE to its pixel location

Streaming: ***approximation*** is a friend

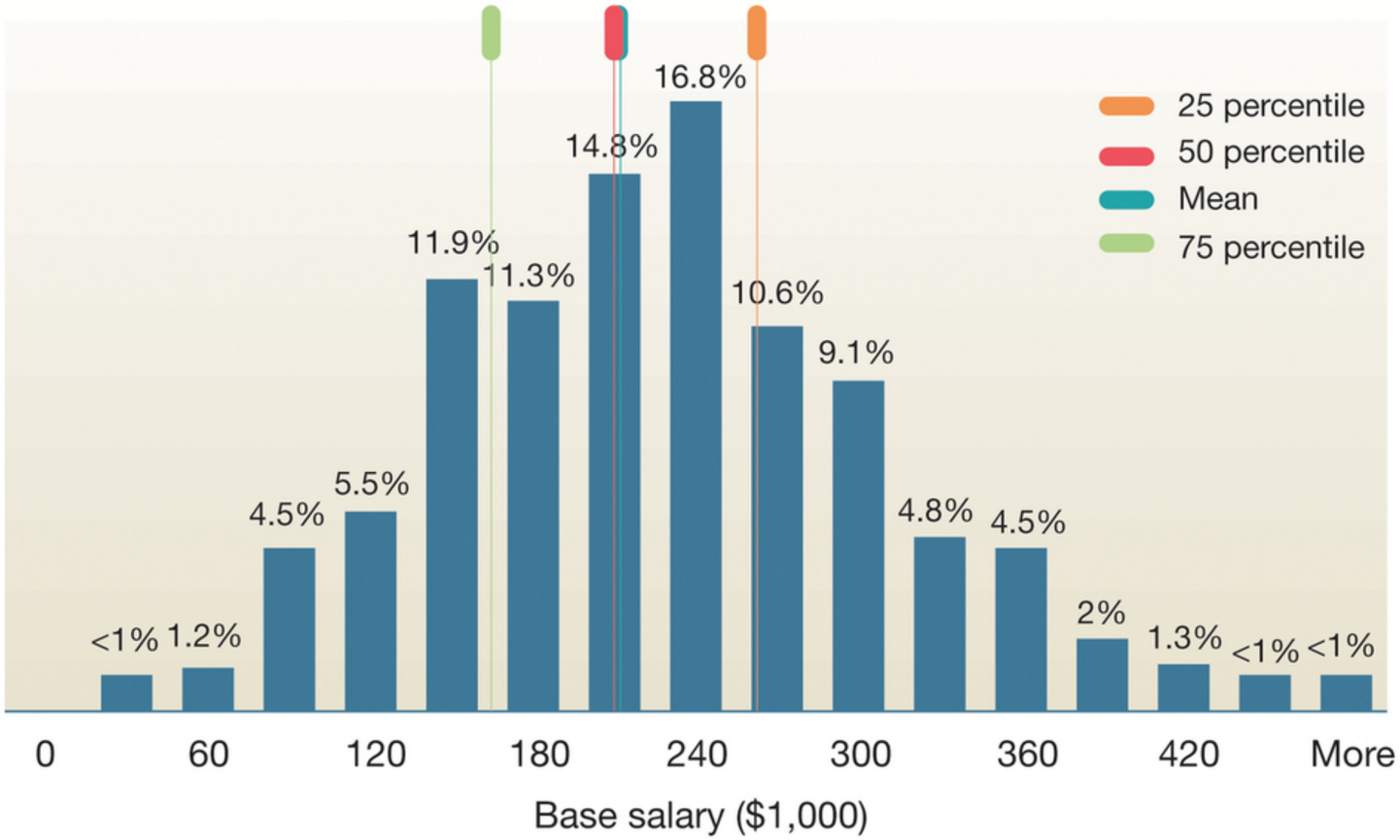
- Need to speed up the computation:
  - The brute force approach often takes a too much time
  - Sacrifice accuracy or granularity of the answers for efficiency
- Common approaches:
  - **Memoization** for *repeated* queries or computations
  - **Indexing** for quickly retrieving a *small* subset of the data
  - **Data cube** for computing *aggregation*

Example: find the median value

- How can we find the median of a large array with streaming?
  - Why is it a challenge?
- This is a well-studied problem in streaming algorithms, lots of methods to approximate the median value within an error bound: generally complex and need multiple passes through the data.
- Most of these methods work on a general assumption that data are continuous and of large range
- In data science projects, sometime, we're only interested in the median range, e.g what is the median salary (range) of non-founder executives?

Example: find the median **range**

- Build a histogram of the values based on interested value ranges
- Select the 50 percentile bucket
- It's the first bucket that passes the 50% mark!
- It is **$210,000**



[*Nature Biotechnology* **32**, 393–395 (2014)]

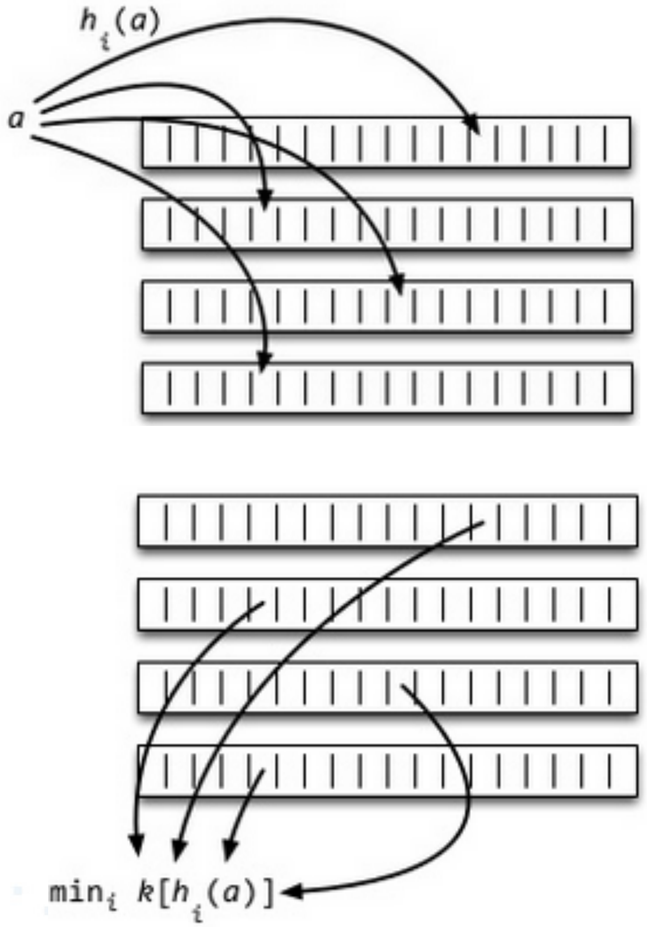
Streaming and Big Data Queries

- Often *exact* answers **not** required
- Results (or a subset of it) can be precomputed
  - In a streaming fashion (right when data arrives)
  - More accurate answers than sampling
  - Can easily be stored on an application server/workstation
- Routinely used in Big Data Analytics at Google, Twitter, Facebook, e.g. Twitter's open source Summingbird toolkit:
  - HyperLogLog (count-distinct) – number of unique users who perform a certain action
  - Count-Min Sketch (hashing) – number of times each query issued to Twitter search
  - Bloom Filters (membership test) – keep track of users who have been exposed to an event

[Boykin et al, VLDB'14]

Summarizing Count-Min Sketch

- Goal: count the frequency of elements in a stream
- Exact solution: requires a large hash table (how?)
- Approximation: use a smaller hash table!
  - There will be collisions
- Count-Min Sketch: use multiple small hash tables!!
  - More tables — Better accuracy



[Cormode and Muthukrishnan, LATIN 2014] and [MapR]

Further Readings on Streaming

- *Data streams: Algorithms and applications*, Muthukrishnan, S. ACM-SIAM Symposium on Discrete Algorithms, 2003. (Background)
- *Selection and sorting with limited storage*, Munro, J. and Paterson, M., Theoretical Computer Science (TCS) 12, 1980. (Median)
- *Probabilistic counting algorithms for data base applications*, Flajolet P. and Martin G., Elsevier JCSS, 31, 1985. (Distinct Element)
- *An Improved Data Stream Summary: The Count-Min Sketch and Its Applications,* Cormode, G. and Muthukrishnan, S., Theoretical Informatics 2004. (Frequency)
- *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm,* Flajolet et al., AofA 2007. (Count Distinct Values).

Data Streaming in Python

Python's **list,** NumPy's **array,** panda's **Series, etc.**

- Compute a function of stream, e.g. average, median, distribution (histograms), distinct elements, etc.

Python's **generators** and **iterators** (on top of **iterables**)

iterable & iterator

- Anything that can be read sequentially in Python is a ***iterable***: dict, string, list, tuple, etc. This is practically anything that can be put inside a **for** loop so that we can sequentially *iterate* through every elements.

```
A = [i for i in xrange(5)]
print 'OUTPUT:',
for i in A:
    print i,
# OUTPUT: 0 1 2 3 4
```

- We can repeatedly iterate through an iterable as many times as we'd like since all the elements are already stored in memory (A is a list)

- Each time you call (aka applying a **for** loop) on an iterable object, it creates a new ***iterator*** automatically to feed the data from memory

generator

- ***generators*** are special classes for creating *iterators* that can only be traversed once.

```
A = (i for i in xrange(5))
print 'OUTPUT:',
for i in A:
    print i,
print '\nOUTPUT:',
for i in A:
    print i,
# OUTPUT: 0 1 2 3 4
# OUTPUT:
```

- generators can be used as data streams as it usually **doesn't store** everything in memory

Example generators

- Python's **file** object: we can only iterate through the file contents **ONCE**

```
with open('filename.csv', 'r') as f:
    for line in f:
        pass
    size = 0
    for line in f:
        size += len(line)
    print 'OUTPUT:', size
# OUTPUT: 0
```

- urllib2.urlopen() returns object where we can only read **ONCE**

Example of non-generators

- Panda's DataFrame, DataSeries
- Numpy's array, matrix
- Python's dict, list, string, pickled objects, etc.

How to create a generator?

- generators are functions BUT use the keyword **yield** in place of **return**
- Example: create a generator of square numbers from 1 to 100
- The code in function body only runs each time the for uses the generator to “generate” data
- The generator is done when the function finishes

```
def square_numbers():
    for i in xrange(10):
        yield i**2

numbers = square_numbers()
print 'FIRST:',
for i in numbers:
    print i,
print '\nSECOND:',
for i in numbers:
    print i,
# FIRST: 0 1 4 9 16 25 36 49 64 81
# SECOND:
```

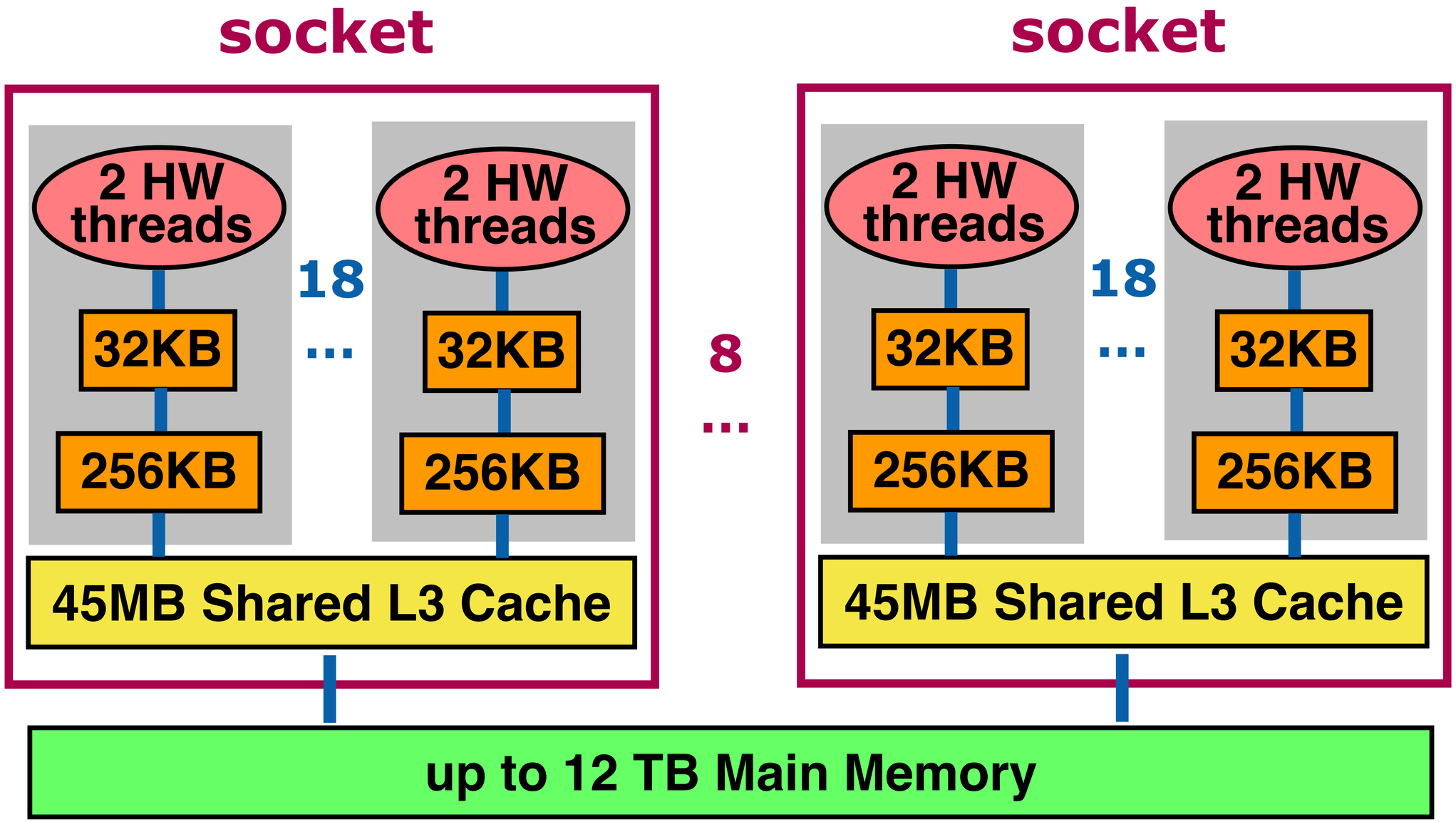
Streaming Summary

- Benefits of Streaming:
  - Reduce the memory footprint
  - Enable pipelining, aka. pushing data out as soon as possible
- Challenges of Streaming:
  - Not every algorithm is streamable
  - Many are not trivial to approximate
- Handling data streams in Python:
  - Generators and Iterators

Big Data Computing Model: Parallelism (more on next class)

- Leverage multi-core and many-core architectures
- Scale up is much faster than Scale out
  - Minimal network communication (slowest in the bandwidth hierarchy)
  - Can fit even several TBs of RAM (compute@cusp has 1TB!)
  - Fast access for spatial (pre-fetch) and temporal (caching) locality
- Hierarchical (non-uniform) memory access matters!

Multicore: 144-core Xeon Haswell E7-v3



Attach: Hard Drives & Flash Devices

Review: General Data Computing Model

- Given a collection of data records (e.g. a large array or list of records)
- Apply a series of *data transformations* to the collection
- Collect the final transformation

birth_year		age		senior (>60)
[1978, 1980, … , 1950]	→	[38, 36, … , 66]	→	[66]

Example: How to do it in Python?

birth_year		age		senior (>60)
[1978, 1980, … , 1950]	→	[38, 36, … , 66]	→	[66]

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

```
birth_year = pd.Series(birth_year)

age = 2016-birth_year

senior = age[age>60]
```

Example: How to do it in Python?

name,birth_year
[('A',1978), … , ('C',1950)] → name,age
[('A',38), … , ('C',66)] → name,senior
(>60 with name 'C')
[('C',66)]

```
age = []
for y in birth_year:
    age.append((y[0], 2016-y[1]))

senior = []
for y in age:
    if y[0]=='C' and y[1]>60:
        senior.append(y)
```

```
birth_year = pd.Series(birth_year)

age = 2016-birth_year

senior = age[age>60]
```

?

The Challenges

- How to specify these *data transformation* so that the system can:
  - bring compute to the data
  - minimize moving or making changes to data
  - operate on collection of data
  - allow users to dictate *what* to be done to the data and LEAVING *how* it will be done to the platform
  - leveraging both scaling up and scaling out capabilities

a declarative and/or functional language for data processing

Higher-Order Functions (Functional)

- Higher-Order functions (functional) are functions that:
  - take functions as arguments
  - **AND/OR** return functions as its results
- HOF is used to abstract common iteration operations
  - *focus on the* **what** *instead of the* **how**
- Using a predefined set of Higher-Order Functions, we can apply data transformation to data using our own **function** (aka transformation)
  - similar to streaming data elements through our function

Example: How to do it i using HOF?

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

Example: How to do it i using HOF?

```
def AGE_TRANSFORM(y):
    return (y[0], 2016-y[1])

def AGE_FILTER(y):
    return y[0]=='C' and y[1]>60


age = map(AGE_TRANSFORM, birth_year)


senior = filter(AGE_FILTER, age)
```

Python's Core HOF

- map(): applies a function over an iterable to produce a new iterable

```
map(int, ['0', '1']) -> [0, 1]
```

- filter(): return only values that satisfy a predicate in the new iterable

```
filter(bool, [0, 1]) -> [1]
```

- reduce(): accumulate a sequence of values from left to right

```
reduce(pow, xrange(5)) -> 0
```

- sorted(): return a new sorted list using a comparator function

Anonymous function: *lambda*

- What if we only use a function once or would like to define it in-place?
- Python's anonymous lambda function:

```
lambda VAR1,VAR2,...,VARN: (expression on VAR1..N)
```

  - No name, no return statement, only an expression to evaluate
  - Come in handy in defining functions in HOF

```
x2 = lambda x: x*x
x2(10) # 100

x_y = lambda x,y: x+y
x_y(1,2) # 3
```

Python's Core HOF

- map(): applies a function over an iterable to produce a new iterable

```
map(lambda x: int(x)+1, ['0', '1']) -> [1, 2]
```

- filter(): return only values that satisfy a predicate in the new iterable

```
filter(lambda x: x<1, [0, 1]) -> [0]
```

- reduce(): accumulate a sequence of values from left to right

```
reduce(lambda x, y: x+y, xrange(5), 100) -> 110
```

- sorted(): return a new sorted list using a comparator function

```
sorted(xrange(5))                          -> [0,1,2,3,4]
sorted(xrange(5), cmp=lambda x,y:y-x) -> [4,3,2,1,0]
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = map(lambda y: 2016-y, birth_year)
```

```
senior = filter(lambda y: y>60, age)
```

Why HOF matters?

- Big Data Platforms including Apache Spark (and the principals of Hadoop) are built on functional programming languages
  - has a smaller set of data processing constructs, thus, easier for the platforms to optimize parallelism
  - less prone to data states and copies
  - keep track of provenance (data + transformation)
- As a user, it keeps us more conscious about concurrency and parallelism
  - algorithm design at a high level (mathematically)

Why HOF matters?

- In a nut shell, programming Big Data Platforms ~ data + a series of Higher-Order functions, where we only have control over our functional arguments:
  - reduce(map(filter(map(A,…),…),…),…)
  - A.map().filter().map().reduce()
- Many of built-in HOF maps well to parallel platforms: data parallelism
  - functions are executed in context independent of each other (except reduce, which has a shared memory buffer)
  - multiprocessing module also provides a parallel version of map()

Questions?

- Big Data Platforms need to abstract data transformation -> Higher-Order functions
  - Think Big Data -> think Functional!
- Next class: Parallel & Distributed Computing, and MapReduce

Good Python Resources on HOF

- https://wizardforcel.gitbooks.io/sicp-in-python/content/
  - “1.6 Higher-Order Functions”
- https://github.com/joelgrus/stupid-itertools-tricks-pydata
  - “Functional Python for Learning Data Science”
  - Slide included